Élection autostabilisante en un nombre polynomial de pas de calcul

Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, Franck Petit

Élection autostabilisante en un nombre polynomial de pas de calcul

Karine Altisen[1], Alain Cournier[2], Stéphane Devismes[1], Anaïs Durand[1] et Franck Petit[3]

[1] *VERIMAG UMR 5104, Université Grenoble Alpes, France*
[2] *MIS Lab., Université Picardie Jules Verne, France*
[3] *LIP6 UMR 7606, INRIA, UPMC Sorbonne Universités, France*

Cet article est un résumé étendu de [1] dans lequel nous présentons un algorithme distribué autostabilisant et silencieux d'élection de leader. Cet algorithme est écrit dans le modèle à états et prouvé sous l'hypothèse d'un démon distribué inéquitable, le démon le plus général du modèle. Il stabilise en $\Theta(n)$ rondes, $\Theta(n^3)$ pas et nécessite $\Theta(\log n)$ bits par processus, où n est le nombre de processus. C'est à notre connaissance le premier algorithme autostabilisant asynchrone d'élection pour lequel une borne supérieure sur le temps de stabilisation en nombre de pas de calcul est prouvée.
Keywords: Algorithmes distribués, tolérance aux pannes, autostabilisation, élection de leader, démon inéquitable

1 Introduction

L'*autostabilisation* [2] est un paradigme permettant de concevoir des systèmes distribués tolérants aux *fautes transitoires*. Les fautes transitoires sont rares, de durée finie et affectent le contenu du composant du réseau où elles surviennent, *e.g.*, la corruption d'une mémoire locale d'un processus ou du contenu d'un message en transit sont deux exemples de fautes transitoires. Après un nombre fini de fautes transitoires et en supposant que ces fautes n'aient pas affecté le code de l'algorithme, un système autostabilisant retrouve de lui-même et en temps fini un comportement correct, c'est-à-dire conforme à sa spécification. On appelle *phase de stabilisation* la période nécessaire au système pour qu'il retrouve un comportement correct. La durée maximale de cette phase est appelée *temps de stabilisation*.

Dans cet article, nous nous intéressons au problème de l'*élection de leader*. Élire un leader dans un réseau consiste à discriminer un unique processus comme étant le *leader*. Nous considérons ici des réseaux connexes identifiés pour lesquels on étend généralement le problème en demandant que tous les processus calculent aussi l'identifiant du leader. Ce problème est une brique de base nécessaire à la conception de nombreux autres protocoles, comme par exemple la construction d'arbres couvrants ou le consensus.

État de l'art. Il existe déjà plusieurs solutions autostabilisantes pour le problème d'élection. Nous nous intéressons ici aux algorithmes écrits dans le *modèle à états* [2] pour lequel l'asynchronisme du système est modélisé par un adversaire, le *démon*. Il existe plusieurs sortes de démons, le démon le plus général étant le *démon distribué inéquitable*. Ce démon ne fait aucune hypothèse d'équité quant à l'activation des processus, ce qui n'est pas le cas d'autres sortes de démons comme par exemple le *démon faiblement équitable*. Le temps de stabilisation et la mémoire nécessaire par processus sont deux des principales mesures de performance d'un algorithme autostabilisant. Dans le modèle à états, le temps de stabilisation peut se mesurer en nombre de *pas de calcul* et en nombre de *rondes*. Informellement, cette dernière unité de mesure permet d'évaluer le temps de stabilisation en fonction de la vitesse du processus le plus lent.

De plus, nous nous intéressons ici aux algorithmes *silencieux*. Un algorithme silencieux converge vers une configuration dite *terminale* où les valeurs des variables de communication des processus ne changent plus. Notez que Dolev *et al.* ont montré [3] qu'un algorithme d'élection silencieux nécessite $\Omega(\log n)$ bits par processus, où n est le nombre de processus.

Les solutions présentées dans [4, 5] nécessitent une mémoire plus importante. La solution d'Arora et Gouda [4] fonctionne sous l'hypothèse d'un démon faiblement équitable et suppose que les processus ont connaissance d'une borne supérieure N sur le nombre de processus. Il converge en $O(N)$ rondes et nécessite

$\Theta(\log N)$ bits par processus. Datta *et al.* [5] proposent la première solution qui fonctionne sous l'hypothèse d'un démon distribué inéquitable. Cet algorithme converge en $O(n)$ rondes. Néanmoins, la complexité en mémoire de cette solution n'est pas bornée.

Les solutions proposées dans [6, 7] sont asymptotiquement optimales en mémoire avec $\Theta(\log n)$ bits nécessaires par processus. La solution de Kravchik et Kutten [6] converge en $O(\mathcal{D})$ rondes, où $\mathcal{D}$ est le diamètre du réseau, cependant elle ne fonctionne que dans les systèmes synchrones. La solution de Datta *et al.* [7] fonctionne sous un démon inéquitable et converge en $O(n)$ rondes.

Contributions. Nous proposons un algorithme autostabilisant et silencieux d'élection pour réseaux asynchrones identifiés. Notre algorithme est écrit dans le modèle à états. Il est prouvé sous l'hypothèse d'un démon distribué inéquitable. Notre algorithme est asymptotiquement optimal en mémoire avec $\Theta(\log n)$ bits nécessaires par processus et converge en au plus $3n + \mathcal{D}$ rondes. Plusieurs de ses performances sont du même ordre que celles du meilleur algorithme existant jusqu'à présent sous nos hypothèses, [7]. En particulier, cet algorithme est prouvé sous l'hypothèse d'un démon distribué inéquitable. Par conséquent, il stabilise en un nombre fini de pas (de calcul). Cependant, aucune borne supérieure sur son temps de stabilisation en pas n'était connue jusqu'à présent. Ceci est la principale différence avec notre algorithme. Nous prouvons que notre algorithme stabilise en $\Theta(n^3)$ pas et nous montrons, au contraire, que l'algorithme de Datta *et al.* stabilise en un nombre de pas non-polynomial.

Plan. Dans la section 2, nous décrivons brièvement le modèle à états. Ensuite, nous présentons les idées principales de notre algorithme ainsi que les grandes lignes de son étude de complexité en section 3. Enfin, nous concluons en section 4. L'ensemble des résultats (correction, étude de complexité, comparaison avec [7], *etc.*) est disponible dans le rapport technique [8].

2 Modèle à états

Nous considérons des systèmes distribués bidirectionnels et connexes composés de *n processus*. Chaque processus p peut communiquer avec un sous-ensemble $\mathcal{N}_p$ de processus appelés ses *voisins*. Les processus sont identifiés par des entiers de manière unique.

Chaque processus possède un nombre fini de *variables localement partagées* via lesquelles il communique : le processus peut lire et écrire dans ses propres variables et peut lire les variables de ses voisins. Nous noterons $p.x$ la variable x du processus p. L'*état* d'un processus est défini par la valeur de ses variables. Une *configuration* du système est définie par l'état de chacun de ses processus.

L'exécution d'un algorithme est composée d'une succession de *pas atomiques* de calcul. À chaque pas de calcul, chaque processus détermine en fonction de son état et de celui de ses voisins s'il est *activable*, c'est-à-dire s'il doit modifier ses variables. Un adversaire, le *démon*, choisit ensuite un sous-ensemble non vide de processus activables. En un pas atomique, les processus choisis sont alors *activés* et mettent à jour leurs variables. Nous supposons que les processus sont activés de manière asynchrone sans hypothèse particulière sur l'équité (le démon est *inéquitable*). S'il n'y a pas de processus activables, l'exécution se termine et la dernière configuration est dite *terminale*.

Le temps de stabilisation d'un algorithme peut être exprimé selon deux mesures : en nombre de pas (atomiques) et en nombre de *rondes*. La première ronde se termine dès lors que tous les processus qui étaient continûment activables depuis le début de l'exécution ont exécuté au moins une action. La seconde ronde commence à la fin de la première, *etc.*

3 Algorithme d'élection autostabilisant

Détaillons maintenant le principe de notre algorithme. Dans cet algorithme, nous choisissons d'élire ℓ [†], le processus d'identité minimum. L'idée de base est de propager l'identité minimum dans le réseau tout en construisant un arbre couvrant enraciné en ℓ. Pour cela, chaque processus p a besoin de trois variables : une variable pour stocker l'identité du processus élu, $p.idR$, et deux variables pour représenter l'arbre couvrant, $p.par \in \mathcal{N}_p \cup \{p\}$, le père du processus, et $p.level$, le niveau du processus dans l'arbre.

Nous adoptons maintenant une démarche pas à pas en commençant par présenter une solution non autostabilisante, puis nous expliquons comment la rendre autostabilisante.

†. Nous ne distinguons pas un processus de son identité. Ainsi, selon le contexte, ℓ désigne soit un processus, soit son identité.

Solution non autostabilisante. Par solution non autostabilisante, nous entendons un algorithme dont toute exécution commence par une initialisation particulière de ses variables. Ici, chaque processus p initialise ses variables comme suit : $p.idR \leftarrow p$, $p.par \leftarrow p$ et $p.level \leftarrow 0$, ce qui signifie que chaque processus est racine de son propre arbre. Notons M la valeur minimum des variables idR des voisins du processus p. Lorsque M est plus petit que $p.idR$, p choisit M en rejoignant l'arbre d'un voisin q tel que $q.idR = M$: $p.idR \leftarrow q.idR$, $p.par \leftarrow q$ et $p.level \leftarrow q.level + 1$. Nous appelons cette action Rejoindre. Si plusieurs voisins q vérifient $q.idR = M$, alors p choisit parmi ces processus celui d'identité minimum. En une ronde, l'identité ℓ est propagée aux processus à distance 1 de ℓ, puis en une ronde supplémentaire à distance 2, *etc.* Petit à petit, l'arbre couvrant enraciné en ℓ est construit. Ainsi, le système atteint en $\mathcal{D}$ rondes une configuration terminale dans laquelle un arbre couvrant enraciné en ℓ est défini : ℓ est élu.

Cependant, cet algorithme n'est pas autostabilisant. En effet, après des fautes, le système peut être dans une configuration où les valeurs des variables sont quelconques. En particulier, la variable idR d'un processus peut avoir n'importe quelle valeur entière, y compris une valeur qui n'est pas l'identité d'un processus. C'est ce que nous appelons une *fausse identité*. Supposons qu'il y ait dans le réseau une fausse identité plus petite que ℓ. Soit x la plus petite fausse identité. L'algorithme converge vers une configuration terminale où x est élu, alors que le processus d'identité x n'existe pas.

Supprimer les fausses identités. Pour rendre notre algorithme autostabilisant, il faut donc introduire un mécanisme de suppression des fausses identités. Plus précisément, notre mécanisme va supprimer toutes les fausses identités plus petites que ℓ.

Tout d'abord, il faut détecter les inconsistances, c'est-à-dire des situations qui ne peuvent pas se produire dans le fonctionnement normal de l'algorithme. Ainsi une racine (normale) r, c'est-à-dire un processus r tel que $r.par = r$, doit satisfaire la condition suivante : $r.idR = r \wedge r.level = 0$. De même, un processus non racine f de père p, c'est-à-dire un processus f tel que $f.par = p \neq f$, doit satisfaire les conditions suivantes : (1) $f.idR$ doit être supérieur ou égal à $p.idR$ et inférieur à sa propre identité (notez que $f.idR$ peut ne pas être égal à $p.idR$ à cause de la situation suivante : p modifie $p.idR$ et ce changement n'est pas encore répercuté par f). (2) Si $f.idR = p.idR$, alors $f.level$ doit être égal à $p.level + 1$.

Ces deux conditions définissent ce qu'est une « bonne relation » entre chaque processus p et le processus désigné par $p.par$. Un processus qui viole cette bonne relation est appelé *racine anormale*. Soit p un processus. Considérons le chemin de taille maximale depuis p, définit par les pointeurs par et tel que tout processus le long de ce chemin satisfait cette bonne relation. Ce chemin est fini et se termine par une racine r, normale ou anormale. Nous disons que p est dans l'arbre de r et nous appelons *arbre anormal* un arbre ayant une racine anormale.

On pourrait penser que pour stabiliser, il suffit à une racine anormale p d'exécuter l'action Réinitialiser suivante : $p.idR \leftarrow p$, $p.par \leftarrow p$ et $p.level \leftarrow 0$. Mais cette méthode ne résout le problème que localement car elle n'empêche pas l'arbre anormal de s'étendre. La fausse identité peut donc continuer à se propager et revenir plus tard sur p. On peut ainsi obtenir une exécution qui ne converge pas. Il faut d'abord s'assurer qu'un arbre anormal ne peut plus grandir avant de le supprimer (avec des actions Réinitialiser).

Pour cela, nous utilisons une variable supplémentaire $status$ qui peut prendre trois valeurs : C pour Clean, EB pour Error Broadcast et EF pour Error Feedback. Nous contraignons un processus p à n'exécuter l'action Rejoindre que si $p.status = C$ et seulement pour rejoindre un voisin de statut C. Ainsi, un arbre ne contenant aucun processus de statut C ne peut plus grandir. On dit alors qu'il est *mort* (sinon, il est dit *vivant*). Le but est donc de faire passer tous les processus des arbres anormaux à un statut différent de C, puis de supprimer ces arbres. Cela nécessite trois phases. Lorsqu'un processus détermine qu'il est une racine anormale, il prend le statut EB. Cette valeur EB est ensuite propagée vers le bas dans tout l'arbre. Lorsqu'une feuille a le statut EB, elle prend le statut EF. Ensuite, les statuts EF sont propagés vers le haut jusqu'à la racine anormale de telle manière qu'au moment où la racine anormale prend le statut EF, tout son arbre a le statut EF. Par conséquent, son arbre est mort. Elle peut donc exécuter l'action Réinitialiser à laquelle on ajoute $p.status \leftarrow C$: une fois réinitialisé, un processus est de nouveau disponible pour l'élection. Suite à une réinitialisation, les anciens fils de la racine deviennent racine d'arbres morts à leur tour, *etc.* Ainsi, les réinitialisations sont propagées vers le bas jusqu'à ce que l'arbre ait disparu.

Lorsque tous les arbres anormaux ont disparus, il n'y a plus de fausses identités plus petites que ℓ dans

le réseau. Notez qu'après ces corrections, il peut rester des fausses identités plus grandes que ℓ, cependant elles ne gênent pas la propagation de ℓ. L'algorithme reprend donc son fonctionnement normal (comme dans la solution non autostabilisante) et finalement ℓ est élu.

La variable supplémentaire *status* peut également être affectée par une faute. Par conséquent, nous introduisons de nouvelles contraintes portant sur cette variable dans la propriété de « bonne relation ». Par exemple, un processus f de statut C ne peut pas avoir un père p de statut EF car le statut EF se propage uniquement de bas en haut. De même, si f a le statut EB, alors p doit avoir le statut EB. Enfin, si f a le statut EF, p doit avoir un statut différent de C.

Temps de stabilisation en rondes. La suppression des arbres anormaux se fait en 3 phases : la propagation de EB, EF et C. Ensuite, nous avons prouvé qu'aucun arbre anormal vivant ne peut être créé s'il n'y a plus de fautes. Enfin, le plus long chemin dans un arbre contient au plus n processus. D'où, chaque phase de la suppression nécessite n rondes, soit $3n$ rondes au total. Une fois la suppression des arbres anormaux terminée, l'arbre couvrant est construit et ℓ est élu en au plus $\mathcal{D}$ rondes. Par conséquent, notre algorithme stabilise en au plus $3n+\mathcal{D}$ rondes.

Temps de stabilisation en pas. À nouveau, puisqu'aucun arbre anormal vivant n'est créé, il y a au plus n arbres anormaux contenant au plus n processus. Chaque processus d'un arbre anormal réalise au plus trois actions pendant la suppression de cet arbre ($C \mapsto EB$, $EB \mapsto EF$ et $EF \mapsto C$). Il y a donc au plus $3n^2$ actions de suppression.

Entre deux suppressions d'arbres anormaux, un processus p peut exécuter des actions Rejoindre. Cependant, chacune de ces actions fait strictement décroître $p.idR$. Il y a au plus n valeurs d'idR différentes dans le réseau entre deux suppressions, donc chaque processus exécute au plus $n-1$ fois Rejoindre entre deux suppressions. Comme il y a au plus n arbres anormaux à supprimer, il y a au plus $n+1$ séquences d'actions Rejoindre consécutives par processus. Il y a donc $O(n^3)$ actions Rejoindre exécutées.

D'où, notre algorithme stabilise en $O(n^3)$ pas.

4 Conclusion

Par manque de place, de nombreux résultats sont omis de cet article. En particulier, nous avons réalisé une étude plus fine de la complexité de notre algorithme. Par exemple, nous avons montré que la borne de $3n+\mathcal{D}$ rondes pour le temps de stabilisation est atteignable. Ensuite, nous avons prouvé des bornes inférieure et supérieure précises sur le temps de stabilisation en pas de calcul : $\frac{n^3}{6}+\frac{3}{2}n^2-\frac{8}{3}n+2$ et $\frac{n^3}{2}+2n^2+\frac{n}{2}+1$.

De plus, nous avons étudié finement la complexité de l'algorithme de Datta *et al.* [7], par exemple nous avons prouvé une borne inférieure en $\Omega((2n)^{\frac{1}{4}\log_2(2n)})$ sur son temps de stabilisation en pas de calcul.

Références

[1] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing Leader Election in Polynomial Steps. In *SSS*, pages 106–119, 2014.

[2] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11) :643–644, 1974.

[3] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization. *Acta Inf.*, 36(6) :447–462, 1999.

[4] Anish Arora and Mohamed G. Gouda. Distributed Reset. *IEEE Trans. Computers*, 43(9) :1026–1038, 1994.

[5] Ajoy Kumar Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing Leader Election in Dynamic Networks. In *SSS*, pages 35–49, 2010.

[6] Alex Kravchik and Shay Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. In *DISC*, pages 91–105, 2013.

[7] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing Leader Election in Optimal Space under an Arbitrary Scheduler. *Theor. Comput. Sci.*, 412(40) :5541–5561, 2011.

[8] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing Leader Election in Polynomial Steps. Technical report, 2014. `hal.archives-ouvertes.fr/hal-00980798`.